Python Language Mapping Specification

September 2002
Version 1.1
formal/02-09-07

An Adopted Specification of the Object Management Group, Inc.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents & Specifications, Report a Bug/Issue.

Contents

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG’s objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

The Open Group

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.

The mission of The Open Group is to drive the creation of boundaryless information flow achieved by:

- Working with customers to capture, understand and address current and emerging requirements, establish policies, and share best practices;
- Working with suppliers, consortia and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies;
- Offering a comprehensive set of services to enhance the operational efficiency of consortia; and
- Developing and operating the industry's premier certification service and encouraging procurement of certified products.

The Open Group has over 15 years experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes tests for CORBA, the Single UNIX Specification, CDE, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industry-standard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in development and quality assurance.

More information is available at http://www.opengroup.org/ .

About CORBA Language Mapping Specifications

The CORBA Language Mapping specifications contain language mapping information for the several languages. Each language is described in a separate stand-alone volume.

Alignment with CORBA

This language mapping is aligned with CORBA, v2.3.1.

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.

- *CORBA Services: Common Object Services Specification* contains specifications for OMG's Object Services.
- *CORBA Common Facilities*: contains services that many applications may share, but which are not as fundamental as the Object Services.
- CORBA domain specifications are comprised of stand-alone documents for each specification; however, they are listed under the domain headings, such as Telecoms, Finance, Med, etc.

OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote.

To obtain OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Definition of CORBA Compliance

The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users aren't required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the *CORBA* specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding specified in this manual.

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to the *Common Object Request Broker: Architecture and Specification, Interworking Architecture* chapter.

As described in the *OMA Guide*, the OMG's Core Object Model consists of a core and components. Likewise, the body of *CORBA* specifications is divided into core and component-like specifications. The structure of this manual reflects that division.

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

Courier bold - Programming language elements.

Helvetica - Exceptions

If applicable, terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Acknowledgments

The following companies submitted parts of this specification:

- GMD Fokus
- Humboldt-Universität zu Berlin
- DSTC Pty Ltd

Python Language Mapping 1

Contents

This chapter contains the following sections.

Source Document(s)

This revision is based on the RTF/FTF report (ptc/01-03-05).

1.1 Mapping Overview

The mapping of IDL to Python presented here does not prescribe a specific implementation. It follows the guidelines presented in Chapter 1.1 of the C Language Mapping (formal/99-07-39). The Python language features used in this mapping are available since Python 1.3, most of them have been around much longer.

This document covers the following aspects of implementing CORBA-based architectures in Python:

- Representation of IDL types, constants, and exceptions in Python
- Invocation of methods on a CORBA object using a statically generated stub
- Invoking methods dynamically (DII)
- Providing object implementations using generated stubs
- Providing object implementations dynamically (DSI)
- Access to ORB services

Issue # 3713- remove the next sentence

~~For some of the concepts, alternative mappings are given. An implementation should clearly identify if it uses these alternative mappings.~~

An implementation of this specification provides the predefined module CORBA. All names qualified with the CORBA module are also provided by the implementation.

1.2 Using Scoped Names

Python implements a module concept that is similar to the IDL scoping mechanisms, except that it does not allow for nested modules. In addition, Python requires each object to be implemented in a module; globally visible objects are not supported[1].

Because of these constraints, scoped names are translated into Python using the following rules:

- An IDL module mapped into a Python module. Modules containing modules are mapped to packages (i.e., directories with an `__init__` module containing all definitions excluding the nested modules). An implementation can chose to map top-level definitions (including the module CORBA) to modules in an implementation-defined package, to allow concurrent installations of different CORBA runtime libraries. In that case, the implementation must provide additional modules so that toplevel modules can be used without importing them from a package.
- For all other scopes, a Python class is introduced that contains all the definitions inside this scope.
- Other global definitions (except modules) appear in a module whose name is implementation dependent. Implementations are encouraged to use the name of the IDL file when defining the name of that module.

For instance,

1. The `__builtin__` module is globally accessible. However, an application like an IDL-to-Python compiler should not introduce new objects into that module.

```
module M
{
 struct E{
  long L;
 };
 module N{
  interface I{
   void import(in string what);
  };
 };
};
const string NameServer="NameServer";
```

would introduce a module **M.py**, which contains the following definitions:

```
# since M is a package, this appears in M/__init__.py
class E:
  pass #structs are discussed later

# module M/N.py
class I:
  def _import(self,what):
    pass #interfaces are discussed later
```

The string **NameServer** would be defined in another module. Because the name of that module is not defined in this specification, using global definitions except for modules is discouraged.

To avoid conflicts, IDL names that are also Python identifiers are prefixed with an underscore (‘_’). For a list of keywords, see Table 1-1.

Table 1-1 Python keywords

and	assert	break	class	continue
def	del	elif	else	except
exec	finally	for	from	global
if	import	in	is	lambda
not	or	pass	print	raise
return	try	while		

1.3 Mapping for Data

1.3.1 Mapping for Basic Types

Because Python does not require type information for operation declarations, it is not necessary to introduce standardized type names, unlike the C or C++ mappings. Instead, the mapping of types to dynamic values is specified here. For most of the simple types, it is obvious how values of these types can be created. For the other types, the interface for constructing values is also defined. The mappings for the basic types are shown in Table 1-2.

Table 1-2 Basic Data Type Mappings.

OMG IDL	Python
octet	`Integer (<type 'int'>)`
short	`Integer`
long	`Integer`
unsigned short	`Integer`
unsigned long	`Long integer(<type 'long int'>)`
long long	`Long integer (<type 'long int'>)`
unsigned long long	`Long integer`
float	`Floating Point Number (<type 'float'>)`
double	`Floating Point Number`
long double	`CORBA.long_double`
boolean	`Integer`
char	`string of length 1`
wchar	`Wide string of length 1`

For the **boolean** type, two predefined values CORBA.TRUE and CORBA.FALSE are available. Since the **wchar** type currently cannot be represented appropriately in Python, an alternative mapping is possible. For the **long double** type, the following interface must be provided:

- The function **`CORBA.long_double`** creates a new **`long double`** number from a **`floating point`** number.
- The operation **`to_float`** of a **`long double`** number converts it into a **`floating point`** number. For each floating point number **`f`**, **`CORBA.long_double(f).to_float==f`**.
- The long double number has an internal representation that is capable of storing IEEE-754 compliant values, with sign, 31 bits of mantissa (offset 16383), and 112 bits of fractional mantissa. If numeric operations are provided, they offer the precision resulting from this specification.

1.3.2 Mapping for Template and Array Types

Both the bounded and the unbounded string type of IDL are mapped to the Python string type. Wide strings are represented by an implementation-defined type with the following properties:

- For the wide string **X** and the integer **n**, **X[n]** returns the nth character, which is a wide string of length 1.
- **len(X)** returns the number of characters of wide string **X**.
- **CORBA.wstr(c)** returns a wide character with the code point **c** in an implementation-defined encoding.
- **X+Y** returns the concatenation of wide strings **X** and **Y**.
- CORBA.word(CORBA.wstr(c)) == c

The sequence template is mapped to sequence objects (e.g., tuples or lists). Applications should not assume that values of a sequence type are mutable. Sequences and arrays of octets and characters are mapped to the string type for efficiency reasons.

For example, given the IDL definitions

```
typedef sequence<long> LongList;
interface VectorOps{
 long sum(in LongList l);
};
```

a client could invoke the operation

```
print obj.sum([1,2,3])
```

An object implementation of this interface could define

```
...
    def sum(self,l):
      return reduce(operator.add,l,0)
```

Array types are mapped like sequence templates. An application should expect a BAD_PARAM exception if it passes sequences that violate the bounds constraint or arrays of wrong size.

A fixed point type **fixed<foo,bar>** is mapped to a Python type or class with the following interface:

- A constructor expecting an integer or large integer with most foo digits.
- Numeric operators for addition, subtraction, multiplication, and division, both of two fixed point numbers and in combination with integers. A DATA_CONVERSION exception is raised if the operation results in a loss of precision.
- Operations value, precision, and decimals.
  - **Fix.value()** returns an integer or large integer
  - **Fix.precision()** returns **foo**
  - **Fix.decimals()** returns bar
- The class **CORBA.fixed** has a constructor expecting foo, bar, and the value. It is used in the case of anonymous fixed types.

1.3.3 Mapping for Enumeration Types

An enumeration is mapped into a number of constant objects in the name space where the enumeration is defined. An application may only test for equivalence of two enumeration values, and not assume that they behave like numbers.

For example, the definition

```
module M{
 enum color{red,green,blue};
 interface O{
  enum Farbe{rot,gruen,blau};
 };
};
```

introduces the objects

```
import M
M.red,M.green,M.blue,M.O.rot,M.O.gruen,M.O.blau
```

1.3.4 Mapping for Structured Types

An IDL struct definition is mapped into a Python class or type. For each field in the struct, there is a corresponding attribute in the class with the same name as the field. The constructor of the class expects the field values, from left to right.

For example, the IDL definition

```
struct segment { long left_limit; long right_limit };
```

could be used in the Python statements

```
s=segment(-3, 7)
print s.left_limit,s.right_limit
```

1.3.5 Mapping for Union Types

Union types are mapped to classes with two attributes. The first is the discriminant **_d**, the second the associated value **_v**. For each branch, there is an additional attribute, which can only be accessed if the branch has been set. There are three possibilities:

- If the discriminant was explicitly listed in a case statement, the value is of the branch associated with that case.
- If the discriminant is not explicitly listed and there is a default case label, the value is of the branch associated with the default case label.
- If the discriminant is not listed, and there is no default, the value is **None**.

The constructor of that class expects the discriminator and the value as arguments.

Alternatively, the union can also be constructed by passing a keyword argument, with the field name of the union as the key. If more than one discriminator is associated with a field, the discriminator must be set explicitly.

For example, the definition

```
union MyUnion switch(long){
 case 1: string s;
 default: long x;
};
```

can be accessed as

Issue # 3715 - add comment

```
u = MyUnion(17, 42)
# 17 is the discriminator, 42 is the value of x
print u.x
u = MyUnion(s = 'string')
print u._d, u._v
```

1.3.6 Mapping for Constants

An IDL constant definition maps to a Python variable initialized with the value of the constant.

1.3.7 Mapping for Exceptions

An IDL exception is translated into a Python class derived from CORBA.UserException. System exceptions are derived from CORBA.SystemException. Both base classes are derived from CORBA.Exception. The parameters of the exception are mapped in the same way as the fields of a struct definition. When raising an exception, a new instance of the class is created; the constructor expects the exception parameters.

For example, the definition

```
module M{
 interface I{
  exception PermissionDenied{string details;};
  I create(in string name)raises(PermissionDenied);
 };
};
```

could be used caught as

```
from M import I;
try:
  i_copy=my_i.create('SuperUser');
except I.PermissionDenied,value:
  print "Could not create SuperUser:",value.details
```

```
i_copy=None
```

1.3.8 Mapping for TypeCodes

TypeCodes are defined in IDL in the *Interface Repository* chapter of the *Common Object Request Broker: Architecture and Specification* document. As a result, the normal mapping rules apply. In addition, the type code constants defined in the *TypeCodes* section (*Interface Repository* chapter) of the *Common Object Request Broker: Architecture and Specification* document are available as Python variables in the module CORBA, with the names given in the *TypeCode Constants* subsection.

For user-defined types, a function **`CORBA.TypeCode`** can be used to create the type codes. This function expects the repository ID. If creation of the type code fails, **`CORBA.TypeCode`** raises a system exception. The repository ID of a type can be obtained with the function **`CORBA.id`**, passing the object representing the type. Such an object shall be available for every IDL type with a **<scoped_name>**, including names that are not otherwise mapped to a Python construct (such as type aliases). If an invalid object is passed to **`CORBA.id`**, a BAD_PARAM system exception is raised.

Example: To obtain the TypeCode of the **`CosNaming::NamingContext`** interface type, either

```
CORBA.TypeCode("IDL:omg.org/CosNaming/NamingContext:1.0")
```

or

```
CORBA.TypeCode(CORBA.id(CosNaming.NamingContext))
```

could be used. In addition, the ORB operations for creating type code, create_*_tc, are available to create type code values. Even though they are defined in PIDL, they follow the mapping for IDL operations in Python.

1.3.9 Mapping for Any

Because of the dynamic typing in Python, there is no need for a strictly type-safe mapping of the any type as in the C or C++ mappings. Instead, all that needs to be available at run-time is the value and the type code corresponding to the type of the value. Because of the mappings for structured types, there is no need that the values belong to the exact class that would have been generated by the IDL compiler. The only requirement is that the values conform to the interface that the IDL compiler would have provided. An object reference extracted from an Any value must be narrowed before it can be used in an interface-specific operation.

To create an any value, the application invokes **CORBA.Any(typecode,value)**. The resulting object supports two operations, **typecode()** and **value()**.

For example, with the IDL specification

```
module M{
 struct S{
```

```
    short l;
    boolean b;
  };
  interface foo{
    void operate(in any on_value);
  };
};
```

a client could perform the actions

```
import M
class Dummy: pass
#construct value
v=Dummy()
v.l=42
v.b=0
#somehow obtain type code
tc=Corba.TypeCode("M::S")
o=something()  #obtain object reference
o.foo(CORBA.Any(tc,v))
```

1.3.10 Mapping for Value Types

A value type **V** (either concrete and abstract) is mapped to a Python class **V**, which inherits from either the base value type, or from **CORBA.ValueBase**. The state of a value is represented in attributes of the instance representing the value. Operations of the **V** are implemented in a class derived from **V** implementing the value. Value implementations may or may not provide an **__init__** method; if they do provide one, which requires parameters, the registered factory is expected to fill in these parameters.

The null value is represented by **None**.

For a given value type, the **ValueFactory** maps to a class instance with a **__call__** method, which returns a new instance of the value type. Initializer operations of the value type map to methods of the factory. The registry for value factories can be accessed using the standard ORB operations **register_value_factory**, **unregister_value_factory**, and **lookup_value_factory**. For value types without operations, a default factory is registered automatically.

If a value type supports an interface (either concrete or abstract), the implementation of the value type can also be supplied as a servant to the POA.

Value boxes are mapped as a Python class with an instance attribute **_boxed**. Instances of the value box are created by passing the boxed value to the constructor of the class.

A **custom** value type inherits from **CORBA.CustomMarshal**, instances need to provide the custom **marshal** and **unmarshal** methods as defined by **CORBA::CustomMarshal**. The types **CORBA::DataOutputStream** and **CORBA::DataInputStream** follow the mapping for abstract values.

1.4 Client Side Mapping

1.4.1 Mapping for Objects and Operations

A CORBA object reference is represented as a Python object at run-time. This object provides all the operations that are available on the interface of the object. Although this specification does not mandate the use of classes for stub objects, the following discussion uses classes to indicate the interface.

The nil object is represented by **None**.

If an operation expects parameters of the IDL Object type, any Python object representing an object reference might be passed as actual argument.

If an operation expects a parameter of an abstract interface, either an object implementing that interface, or a value supporting this interface may be passed as actual argument. The semantics of abstract values then define whether the argument is passed by value or by reference.

Operations of an interface map to methods available on the object references. Parameters with a parameter attribute of **in** or **inout** are passed from left to right to the method, skipping **out** parameters. The return value of a method depends on the number of **out** parameters and the return type. If the operation returns a value, this value forms the first *result value*. All inout or out parameters form consecutive *result values*. The method result depends then on the number of *result values*:

- If there is no *result value*, the method returns None.
- If there is exactly one *result value*, it is returned as a single value.
- If there is more than one *result value*, all of them are packed into a tuple, and this tuple is returned.

Assuming the IDL definition

```
interface I{
 oneway void stop();
 bool more_data();
 void get_data(out string name,out long age);
};
```

a client could write

```
names={}
while my_I.more_data():
  name,age = my_I.get_data()
  names[name]=age
```

my_I.stop()

If an interface defines an **attribute name**, the attribute is mapped into an operation **_get_name**, as defined. If the attribute is not **readonly**, there is an additional operation **_set_name**, as defined in the *OMG IDL Syntax and Semantics* chapter, "Attribute Declaration" section, of the *Common Object Request Broker: Architecture and Specification* document.

1.4.2 Narrowing Object References

Python objects returned from CORBA operations or pseudo-operations (such as **string_to_object**) might have a dynamic type, which is more specific than the static type as defined in the operation signature.

Since there is no efficient and reliable way of automatically creating the most specific type, explicit narrowing is necessary. To narrow an object reference **o** to an interface class **I**, the client can use the operation **o._narrow(I)**.

Implementations may give stronger guarantees about the dynamic type of object references.

1.4.3 Mapping for Context

The Context object supports the following operations:

- set_one_value(name,val) associates a property name with a property value.
- set_values(dict) sets a number of properties, passed as a dictionary.
- get_values(prop_name,start_scope=None) returns a dictionary of properties that match with prop_name. If the key word argument start_scope is given, search is restricted to that scope.
- delete_values(prop_name) deletes the specified properties from the context.
- create_child(ctx_name) returns a new child context.

All property names and values are passed as strings. Instead of returning Status values, these operations may raise CORBA system exceptions.

1.4.4 The Dynamic Invocation Interface

Because Python is not statically typed, there is no need to use the NVList type to pass parameters at the DII. Instead, the **_create_request** operation takes the parameters of the operation directly.

The operation **_create_request** of **CORBA.Object** instances returns a **Request** object and takes the following parameters:

- the name of the operation
- a variable list of parameters
- optionally the keyword argument **context**

- optionally the keyword argument **`flags`**
- optionally the keyword argument **`repository_id`**

The parameters are passed following the usual conventions for values of their respective types. It is the responsibility of the run-time system to correlate these values to the types found in the interface repository. The application may specify the repository id of the target object. Instead of returning a Status value, **`_create_request`** might raise a CORBA system exception.

The resulting Request object supports the following operations:

- **`invoke(flags=0)`** synchronously initiates the operation.
- **`send(flags=0)`** asynchronously initiates the operation.
- **`get_response(flags=0)`** can be used to analyze the status of the operation. This returns the result value and out parameter, and may raise both user and system exceptions.
- **`delete(flags=0)`** can be used to invalidate a request.

The various flags defined in the **`CORBA`** module follow the normal mapping rules. Some of the flags deal with memory management and have no specified semantics in Python. Relevant to the DII are the following flags: **`INV_NO_RESPONSE`**, **`INV_TERM_ON_ERR`**, and **`RESP_NO_WAIT`**.

1.4.5 Mapping for Components

The CORBA Component specification defines a number of new IDL Syntax elements. It also explains how these syntax elements result in implicit interface definitions, with implicit operations. A component-aware Python program should use the implicit operation names to access the component.

1.5 Server Side Mapping

Traditionally, IDL language mapping would be unspecific on purpose when it comes to a mapping for object implementations. The reasoning was that there are various reasonable approaches, and standardizing on a single approach would limit the range of applications.

Central to the architecture is the object adapter, which communicates the requests to the implementation. CORBA explicitly allows for multiple object adapters, including non-standardized ones. The only object adapter that has been standardized for CORBA 2.0 is the Basic Object Adapter (BOA), as a least common denominator. This adapter has been found to be insufficient, so vendors would extend it with proprietary features.

A recent effort was made to standardize a portable object adapter (POA). The POA standard [BDE97] now suggests to drop the BOA from the *Common Object Request Broker: Architecture and Specification*, and replace it with the POA (note: this occurred in Version 2.2 of the *Common Object Request Broker: Architecture and Specification*). Vendors are still free to support other object adapters, including the old BOA.

This specification only defines a server side mapping for the POA. Many of the relevant definitions are defined using IDL in [BDE97]. The corresponding Python mapping follows the rules specified above.

1.5.1 Skeleton-Based Implementation

Issue # 3719 - clarify text in the next paragraph

~~One approach of implementing interfaces is to derive the implementation class from a skeleton class.~~ This specification defines an inheritance-based mapping for implementing servants. Delegation-based approaches are also possible, but can be implemented on top of the inheritance-based approach. For the POA, the first element of the fully-scoped name of the interface is suffixed with "__POA". Following the name mapping scheme for Python, the corresponding Python class can be used as a base class for the implementation class. For example, the interface

```
module M{
 interface I{
  void foo();
 };
};
```

could be implemented in Python as

```
import M__POA
class MyI(M__POA.I):
  def foo(self):
    pass #....
```

If the implementation class derives from other classes that also implement CORBA interfaces, the skeleton class must be mentioned before any of those base classes. A class may implement multiple interfaces only if these interfaces are in a strict inheritance relationship.

The skeleton class (`M__POA.I` in the example) supports the following operations:

- `_default_POA()` returns the POA reference that manages that object. It can be overridden by implementations to indicate they are managed by a different POA. The standard implementation returns the same reference as `ORB.resolve_initial_reference("RootPOA")`, using the default ORB.
- `_this()` returns the reference to the object that a servant incarnates during a specific call. This works even if the servant incarnates multiple objects. Outside the context of an operation invocation, it can be used to initiate the implicit activation, if the POA supports implicit activation. In any case, it should return an object that supports the operations of the corresponding IDL interface.

The base class for all skeleton classes is the class `PortableServer.Servant`.

1.5.2 The Dynamic Skeleton Interface

An implementation class is declared as dynamic by inheriting from **PortableServer.DynamicImplementation**. Derived classes need to implement the operation **invoke**, which is called whenever a request is received. The PIDL type **ServerRequest** is not mapped to a structure, but to a parameters list for that operation. **invoke** is passed the following parameters:

- the name of the operation.
- a variable list of parameters, following the usual mapping rules for the parameter types of the specified operation.
- a keyword parameter **context**, specifying the context object if any, or **None**.

invoke returns either with a result following the mapping for out parameters, or by raising an appropriate exception.

The implementation class must also implement the pseudo-operation **_get_interface**, which must return a non-nil **CORBA::InterfaceDef** reference. It does not need to implement any other pseudo operation.

1.5.3 Mapping for the Cookie Type

Because the Cookie type is a **native** type, a Python mapping is required:

```
class Cookie: pass
```

According to the language mapping, the **preinvoke** operation of the **ServantLocator** returns a tuple (servant, cookie). The cookie will be input later to the **postinvoke** operation. The **ServantLocator** implementation is free to associate any attributes with the cookie.

1.5.4 Mapping for Components

A component implementation consists of a set of interface implementations. The names of these interfaces are defined in the Components specification; these interfaces follow the standard mapping rules for interfaces in Python. This specification does not define a mapping of the Component Implementation Framework to Python.

1.6 Mapping for ORB Services

The predefined module CORBA contains the interfaces to the ORB services. The first step that needs to be performed is the ORB initialization. This is done using the **ORB_init** operation:

```
orb=CORBA.ORB_init(argv,orbid)
```

Both the argument vector and the orbid are optional. If provided, the orbid must be a string, and the argument vector must be similar to **sys.argv**. If no orbid is given, the default ORB object is returned.

Depending on the object adapters provided, the ORB object may provide additional initialization functions. Furthermore, two operations allow access to the initial references:

1. **orb.list_initial_references()** returns a list of names of available services.
2. **orb.resolve_initial_reference(string)** returns an object reference of raises ORB_InvalidName.

Two operations are available for stringification of object references:

1. **orb.string_to_object(string)** returns an object reference, or a nil reference if the string is not understood.
2. **orb.object_to_string(object)** returns a stringification of the object reference that can be passed later to **string_to_object**.

Each object reference supports a number of operations:

- **_get_implementation()** returns an **ImplementationDef** object related to the object.
- **_get_interface()** returns an **InterfaceDef** object.
- **_is_a(string)** expects a repository identifier and returns true if the object implements this interface.
- **_non_existent()** returns true if the ORB can establish that the implementation object behind the reference is gone.
- **_hash(maximum)** returns a value between 0 and maximum that does not change in the lifetime of the object.
- **_is_equivalent(other_object)** returns true if the ORB can establish that the references reference the same object.

The interface ORB provides some additional functions:

- **get_default_context()** returns the default context
- **send_multiple_requests_oneway**, **send_multiple_requests**, **get_next_response**, and **poll_next_response** are used with the DII.

1.7 Deprecated Interfaces

Because some interfaces and operations of earlier CORBA specifications are deprecated in the *Common Object Request Broker: Architecture and Specification* (CORBA 2.2), no mapping is provided for these interfaces:

- **get_current()**. Applications should use **resolve_initial_reference** instead.

- **get_implementation()** and the **ImplementationDef** interface, as well as the mapping for the Basic Object Adapter. Applications should use the Portable Object Adapter.
- **get_principal** and the **Principal** interface. Applications should use **SecurityLevel2::Credentials** instead.

Python Language Mapping, v1.1
Reference Sheet

This is a revised version of the Python Language Mapping specification.

OMG document(s) used to create this version:

- FTF Final Report: ptc/01-03-05